



Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencias de la Computación

Introducción a la Programación IIC1103

Prof. Ignacio Casas
icasas@ing.puc.cl

Tema 10 – Recursión Parte 2

Colaboración de Mauricio Arriagada

Agenda

- (1) Retorno sucesivo en una Recursión.
Repaso de funciones:
Invocación a funciones, suspensión del flujo y retorno
Funciones “Nones”
- (2) Recursión simple y “Tail Recursion”
- (3) Back-Tracking
- (4) Ejemplos

Repaso: Retorno sucesivo en una Recursión

Cuando construimos un **algoritmo recursivo**, hacemos que una función **se invoque a si misma** un cierto número de veces hasta que **converge** a un **caso base** (si lo hemos definido correctamente) donde detiene (termina) las invocaciones.

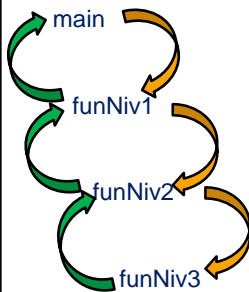
Una vez que ha llegado al **caso base**, ya no se invoca a si misma y debe **retornar/regresar** (el control de la ejecución) a la llamada previa. Y así **retorna** sucesivamente **a cada llamada previa**, realizando en el camino cualquier operación que quedó pendiente (o suspendida).

El concepto de una función (con o sin recursión) que “**retorna**” (el control de la ejecución) al código que la invocó, es un poco difícil de “digerir” cuando se estudia por primera vez. Repasemos este concepto primero con funciones sin recursión.

3

Repaso: Invocación a Funciones (no Recursivas)

Ejemplo de un programa con tres funciones:



```

""" Cómo funcionan las funciones """
def funNiv1(s=""):
    s += " "
    print(s+"Nivel 1: me acaban de invocar desde main")
    print(s+"Nivel 1: voy a invocar al nivel 2")
    funNiv2(s)
    print(s+"Nivel 1: acabo de regresar del nivel 2")
    print(s+"Nivel 1: chao, regreso al main")
def funNiv2(s=""):
    s += " "
    print(s+"Nivel 2: me acaban de invocar desde nivel 1")
    print(s+"Nivel 2: voy a invocar al nivel 3")
    funNiv3(s)
    print(s+"Nivel 2: acabo de regresar del nivel 3")
    print(s+"Nivel 2: chao, regreso al nivel 1")
def funNiv3(s=""):
    s += " "
    print(s+"Nivel 3: me acaban de invocar desde nivel 2")
    print(s+"Nivel 3: me cansé y no voy a invocar a nadie")
    print(s+"Nivel 3: chao, regreso al nivel 2")
# main
print ("main: invoco desde el main a la función Nivel 1")
funNiv1()
print("main: se acabó la ejecución de la función Nivel 1")
print("main: chao")
  
```

4

Repaso: Resultado del programa anterior

```
>>>
main: invoco desde el main a la función Nivel 1
      Nivel 1: me acaban de invocar desde main
      Nivel 1: voy a invocar al nivel 2
            Nivel 2: me acaban de invocar desde nivel 1
            Nivel 2: voy a invocar al nivel 3
                  Nivel 3: me acaban de invocar desde nivel 2
                  Nivel 3: me cansé y no voy a invocar a nadie
                  Nivel 3: chao, regreso al nivel 2
            Nivel 2: acabo de regresar del nivel 3
            Nivel 2: chao, regreso al nivel 1
      Nivel 1: acabo de regresar del nivel 2
      Nivel 1: chao, regreso al main
main: se acabó la ejecución de la función Nivel 1
main: chao
```

Como se
interpreta:

El programa main invoca a la función funNiv1()
 funNiv1() invoca a la función funNiv2()
 funNiv2() invoca a la función funNiv3()
 funNiv3() se ejecuta y retorna (el control) a funNiv2()
 funNiv2() retorna (el control del flujo) a funNiv1()
 funNiv1() retorna (el control del flujo) al programa "main"
 El programa main continúa su ejecución.

5

Repaso: Funciones/Métodos en Python

Las funciones/métodos pueden: ejecutar distintas instrucciones (por ejemplo: ciclos, condicionales, print e input) y luego retornar (al código que las invocó) uno o varios valores a través del comando **return**.

También pueden retornar (al código que las invocó) "nada" ("None") con un **return** sin valores o simplemente si no utilizamos **return**.

A estas funciones las llamamos "**Nones**" o "**Ñoñas**" (para distinguirlas de las que retornan algo).

Ejemplo de funciones/métodos "Nones":

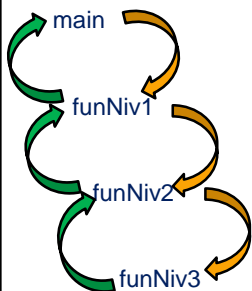
```
def ImprimeMayor(valor1, valor2):
    if valor1 >= valor2:
        print(str(valor1)+" es >= que "+str(valor2))
    else:
        print(str(valor1)+" es < que "+str(valor2))
    return
```

este "return" es opcional

L.append(z)
 L.remove(x)
 L.reverse()
 L.sort()

Repaso: Invocación a Funciones “Nones” (no Recursivas)

Ejemplo de un programa con tres funciones:



```

""" Cómo funcionan las funciones Nones """
def funNiv1(s=""):
    s += " "
    print(s+"Nivel 1: me acaban de invocar desde main")
    print(s+"Nivel 1: voy a invocar al nivel 2")
    a = funNiv2(s)
    print(s+"Nivel 1: acabo de regresar del nivel 2")
    print(s+"Nivel 1: la función nivel 2 retornó: ", a)
    print(s+"Nivel 1: chao, regreso al main")
def funNiv2(s=""):
    s += " "
    print(s+"Nivel 2: me acaban de invocar desde nivel 1")
    print(s+"Nivel 2: voy a invocar al nivel 3")
    a = funNiv3(s)
    print(s+"Nivel 2: acabo de regresar del nivel 3")
    print(s+"Nivel 2: la función nivel 3 retornó: ", a)
    print(s+"Nivel 2: chao, regreso al nivel 1")
def funNiv3(s=""):
    s += " "
    print(s+"Nivel 3: me acaban de invocar desde nivel 2")
    print(s+"Nivel 3: me cansé y no voy a invocar a nadie")
    print(s+"Nivel 3: chao, regreso al nivel 2")
# main
print ("main: invoco desde el main a la función Nivel 1")
a = funNiv1()
print("main: se acabó la ejecución de la función Nivel 1")
print("main: la función nivel 1 retornó: ", a)
print("main: chao")
  
```

Repaso: Resultado del programa anterior

```

>>>
main: invoco desde el main a la función Nivel 1
Nivel 1: me acaban de invocar desde main
Nivel 1: voy a invocar al nivel 2
Nivel 2: me acaban de invocar desde nivel 1
Nivel 2: voy a invocar al nivel 3
Nivel 3: me acaban de invocar desde nivel 2
Nivel 3: me cansé y no voy a invocar a nadie
Nivel 3: chao, regreso al nivel 2
Nivel 2: acabo de regresar del nivel 3
Nivel 2: la función nivel 3 retornó: None
Nivel 2: chao, regreso al nivel 1
Nivel 1: acabo de regresar del nivel 2
Nivel 1: la función nivel 2 retornó: None
Nivel 1: chao, regreso al main
main: se acabó la ejecución de la función Nivel 1
main: la función nivel 1 retornó: None
main: chao
  
```

retorna nada ("None")

El arte de la recursión

La recursión no es un concepto de exclusiva aplicación en matemática o programación.

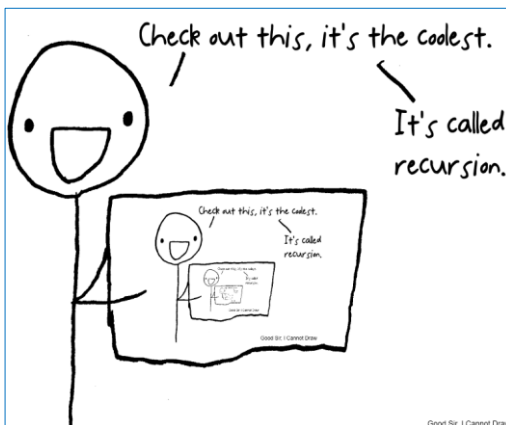
También el mundo de la literatura, el cine o el diseño han explotado la recursión.

El libro de “Las mil y una noches”, por ejemplo, es un relato que incluye relatos que, a su vez, incluyen relatos.

Numerosas películas o series de televisión incluyen en su trama el rodaje de otras películas: “Cantando bajo la lluvia”, de Stanley Donen y Gene Kelly, “Nickelodeon”, de Peter Bogdanovich, o “Vivir rodando”, de Tom DiCillo, incluso “**LOST**”.

Recursión simple (con operaciones pendientes para el regreso)

Tail Recursion (sin operaciones pendientes para el regreso)



El arte de la recursión

Recordemos...

Las soluciones recursivas “simples” involucran 2 partes:

1.- Caso Base:

Este caso debe ser simple de resolver directamente, sin recursión y debe permitir terminar las invocaciones.

2.- Invocación a si mismo:

- (a) **Dividir** el problema en una o mas partes simples y similares.
- (b) **Llamarse a si mismo** (función recursiva) en cada parte.
- (c) **Converger**: Las invocaciones recursivas deben llegar al caso base.
(Cuidado con las recursiones infinitas.)
- (d) **Combinar** la solución de las partes en una solución completa.

11

El arte de la recursión

Recordemos...

```
""" concepto de recursión simple """
def recursion (problema):
    if (problema == caso base):
        return (solución caso base)
    else:
        x = operaciones sobre recursion (subproblema)
        return x
```

Cuando regresa de cada invocación,
se realizan las operaciones “pendientes”.

12

Tail recursion

Si bien la solución recursiva puede ser más elegante y simple que la iteración, presenta una importante desventaja:

utiliza más memoria (que la iteración) para mantener un “stack” (registro) de invocaciones a la función... para poder “volver atrás”.

La “**tail recursion**” intenta corregir esta ineficiencia:

trata que no quede ninguna instrucción/operación “en el camino” y así el proceso pueda terminar efectivamente cuando llega al caso base... sin necesidad de “regresar”... Sin necesidad de mantener un “stack” de invocaciones.

Una función recursiva simple puede convertirse a **tail recursive** usando en la función original un parámetro adicional, para ir guardando el resultado parcial de tal manera que la llamada recursiva ya no deje operaciones pendientes.

Tail recursion

No todos los software (intérpretes o compiladores) de lenguajes tienen la capacidad de detectar una “**tail recursion**” para eliminar el uso del stack de invocaciones y terminar la recursión sin regresar cuando se llega al caso base.

Si bien Python actualmente no tiene esta capacidad, igual es interesante (e inteligente) conocer como se hace. Haremos “**tail recursion**”, pero Python no sacará provecho de esto.

Veamos un **ejemplo** sencillo:

Sumar los elementos de una lista de números en forma recursiva.

Solución iterativa

Ejemplo 1:

Sumar los elementos de una lista de números

Primero...¿Cómo lo resolvemos de forma iterativa???

```
def Suma_Iter(Lista):  
    result = 0  
    for x in Lista:  
        result += x  
    return result
```

15

Solución recursiva

Ejemplo 1:

Sumar los elementos de una lista de números de forma recursiva

1.- Caso Base →Cuál es el caso base?

2.- Caso Recursivo

→ Cómo dividimos el problema?

→ Donde invocamos a la función recursivamente?

→ Finalmente donde se combina la solución ?

16

Solución recursiva

Ejemplo 1:

Sumar los elementos de una lista de números de forma recursiva

```
def Suma_Rec(Lista):
    if Lista == []:
        return 0
    else:
        return Lista[0] + Suma_Rec(Lista[1:])
```

¿Necesariamente debe estar en el **return** la llamada recursiva?
No. Veamos una manera equivalente de hacerlo.

17

Solución recursiva

```
def Suma_Rec(Lista):
    if Lista == []:
        return 0
    else:
        suma = Lista[0] + Suma_Rec(Lista[1:])
        return suma
```

```
print(Suma_Rec([1, 5, 3]))
```

invocaciones {
 rec_suma([1, 5, 3])
 suma = 1 + rec_suma([5, 3])
 suma = 1 + (5 + rec_suma([3]))
 suma = 1 + (5 + (3 + rec_suma([])))
 suma = 1 + (5 + (3 + 0))
 regreso {
 suma = 1 + (5 + 3)
 suma = 1 + 8
 suma = 9

Nota: NO es tail recursion.

18

```

1 print(Suma_Rec([1, 5, 3]))
2 def Suma_Rec(Lista):
3     if Lista == []:
4         return 0
5     else:
6         suma = Lista[0] + Suma_Rec(Lista[1:])
7         return suma
8
9 def Suma_Rec(Lista):
10    if Lista == []:
11        return 0
12    else:
13        suma = Lista[0] + Suma_Rec(Lista[1:])
14        return suma
15
16 def Suma_Rec(Lista):
17    if Lista == []:
18        return 0
19    else:
20        suma = Lista[0] + Suma_Rec(Lista[1:])
21        return suma
22
23 def Suma_Rec(Lista):
24    if Lista == []:
25        return 0
26    else:
27        suma = Lista[0] + Suma_Rec(Lista[1:])
28        return suma

```

19

Solución tail recursion

```

def Suma_Tail(Lista, Total=0):
    if Lista == []:
        return Total
    else:
        return Suma_Tail(Lista[1:], Total + Lista[0])

```

invocaciones	{	Lista = [1,5,3] , Total = 0
		Lista = [5,3] , Total = 0 + 1
		Lista = [3] , Total = 1 + 5
		Lista = [] , Total = 6 + 3
regreso	{	Total = 9
		Total = 9
		Total = 9
		Total = 9

Ahora SI es tail recursion.

Ejemplo 2: exponenciación recursiva

Desafío: Escribir una función que toma un valor **base** y un exponente **exp** y de forma recursiva calcula:

$$base^{exp}$$

Nota: No puedes usar ****** o **math.pow**



21

Ejemplo 2: exponenciación recursiva

Escribir una función que toma un valor **base** y un exponente **exp** y de forma recursiva calcula:

$$base^{exp}$$

```
def RecExpo (base, exp) :
    if exp == 0:
        return 1
    elif exp > 0:
        return base * RecExpo (base, exp-1)
    return RecExpo (base, exp+1) / base
```

Con “tail recursion”:

```
def RecExpoTail (base, exp, result=1) :
    if exp == 0:
        return result
    elif exp > 0:
        return RecExpoTail (base, exp-1, result*base)
    return RecExpoTail (base, exp+1, result/base)
```

¿Cómo podríamos disminuir el número de recursiones?
(independiente del “tail”). **Ayuda:** detectar exponente “par”.

Ejemplo 3: cuenta regresiva recursiva

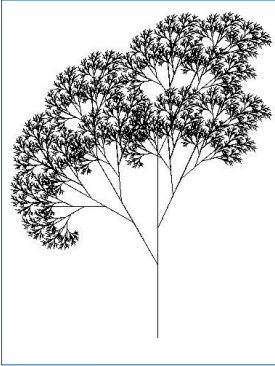
Escribir una función recursiva que imprima números de **n** hasta **0**.

```
def CuentaR(n):  
    """ imprimir números de n hasta cero """  
    print(n)  
    if n == 0:  
        return  
    elif n < 0:  
        return CuentaR(n+1)  
    else:  
        return CuentaR(n-1)  
#  
print(CuentaR(10))  
print(CuentaR(-10))
```

23

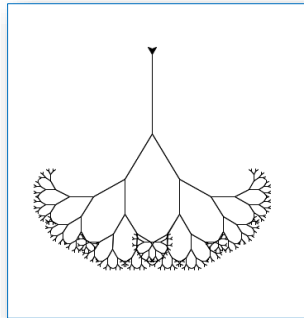
Back-Tracking





Algoritmo de Back-Tracking:

utiliza la recursión para recorrer un árbol invertido en forma ordenada



25

Back-Tracking

El backtracking (método de retroceso ó vuelta atrás) es una técnica general de resolución de problemas, aplicable tanto a problemas de optimización, juegos y otros tipos.

El backtracking realiza una búsqueda exhaustiva y sistemática en el espacio de soluciones. Por ello, suele resultar **muy costoso (en tiempo de ejecución)**.

El back-tracking es "naturalmente" recursivo.

La solución del problema se puede representar como un árbol invertido donde cada nodo representa un problema similar pero más sencillo.

Se debe recorrer cada "rama del árbol".

En una "rama" cualquiera, se termina en un "último" nodo que representa el caso base (más sencillo) de la recursión.

Si el caso base es una solución, se termina el algoritmo.

Si no lo es, se "retorna" al nodo anterior para continuar en otra rama.

26

BACK-TRACKING:

Se deben “recorrer” muchos caminos posibles (en el “árbol invertido”) hasta encontrar la solución o decidir que no hay una solución.

Cada camino (“rama”) se recorre hasta el final: si no hay solución en esa rama, se retorna al nodo (bifurcación) anterior, para continuar con las instrucciones indicadas después del llamado a la función/método recursiva/o.

Se recorren varios caminos (“ramas”) hasta encontrar una solución (peor caso: se recorren todos los caminos posibles).

Back-Tracking

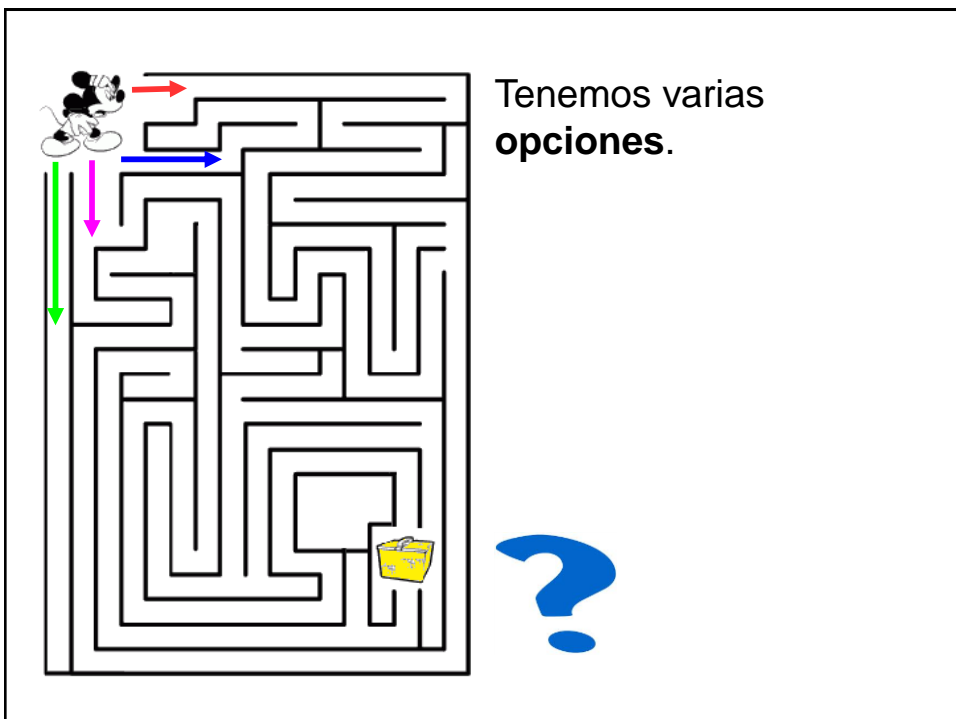
Back-Tracking:

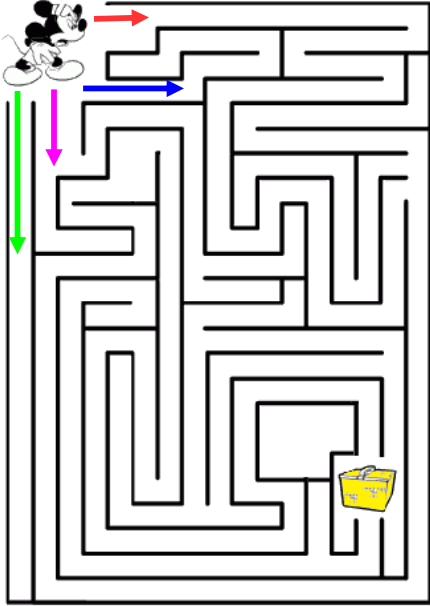
Es una solución **recursiva** para problemas donde se deben **revisar exhaustivamente muchos** (a veces todos) los posibles **caminos**.

Se puede definir la solución del problema como el **recorrido exhaustivo de un “árbol invertido”**, donde cada **nodo** puede ser:

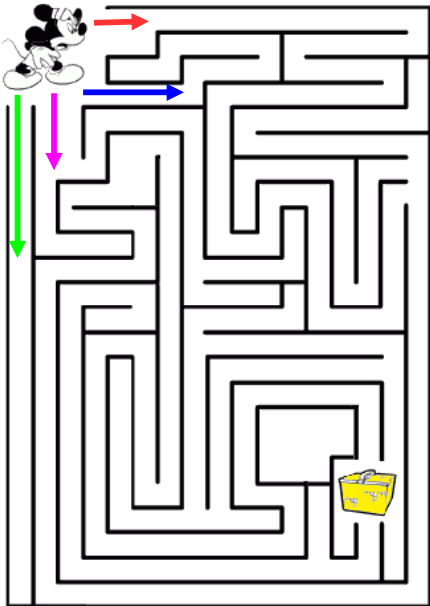
- una **bifurcación** (dos o más posibles alternativas a seguir), o
- el **término** de un camino, o
- una **solución** al problema.

Cuando se llega al término de un camino sin haber encontrado una solución, se debe **regresar** a la bifurcación previa para continuar con la siguiente alternativa. La acción de “**regreso**” es natural para la **recursión**.

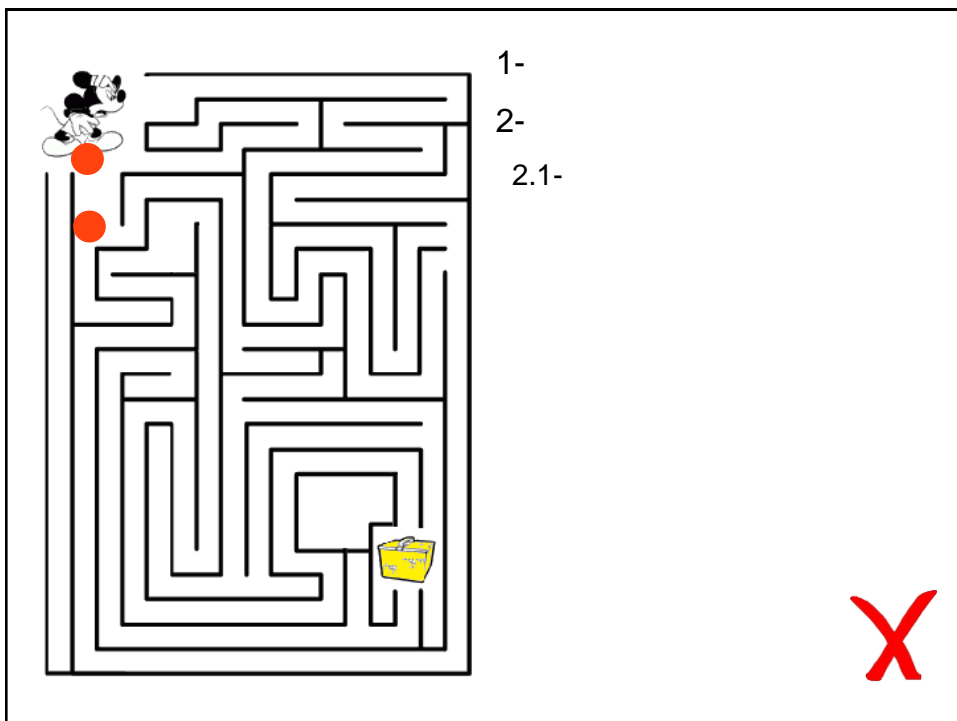
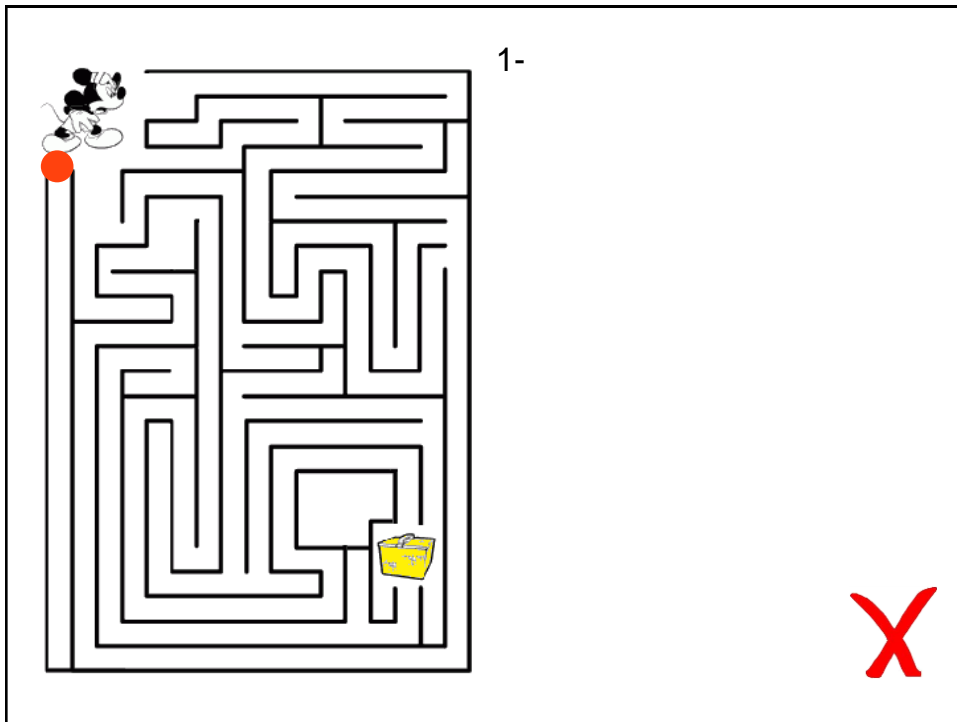


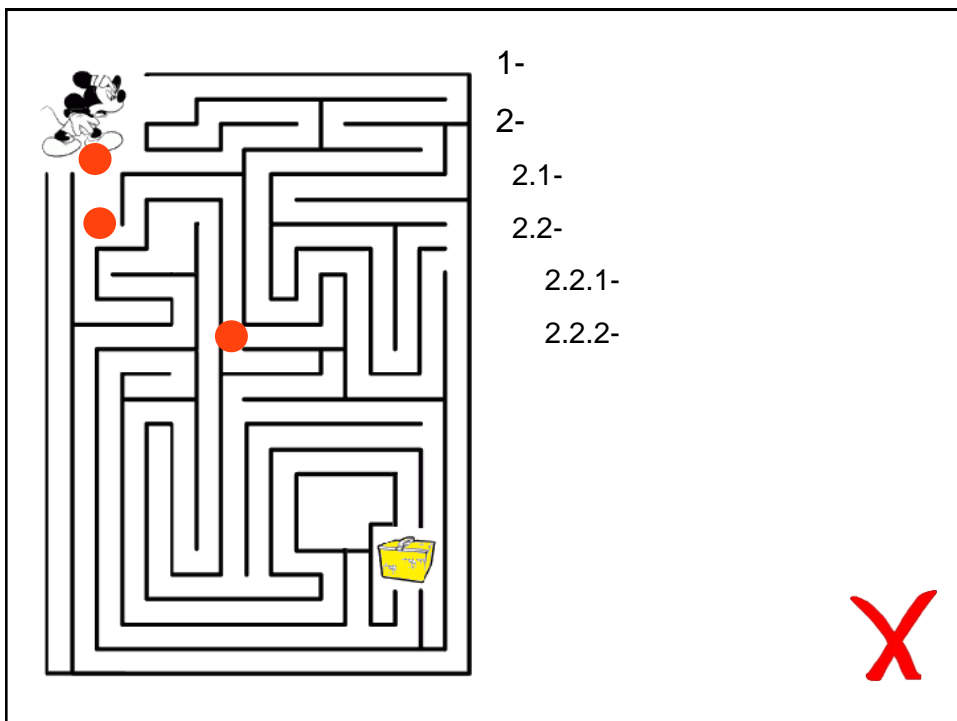
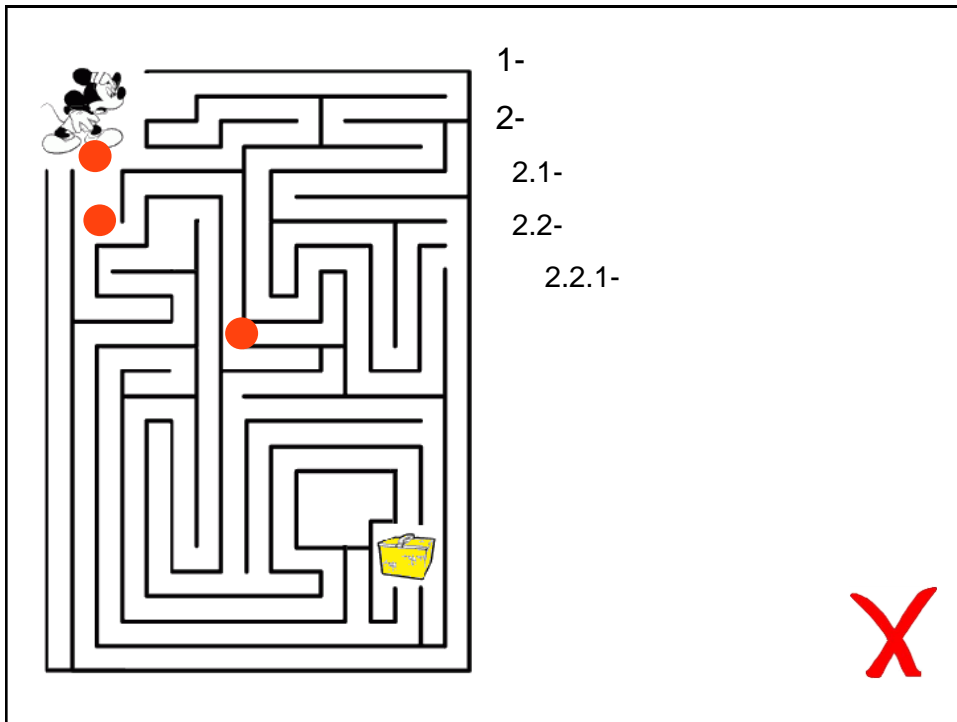


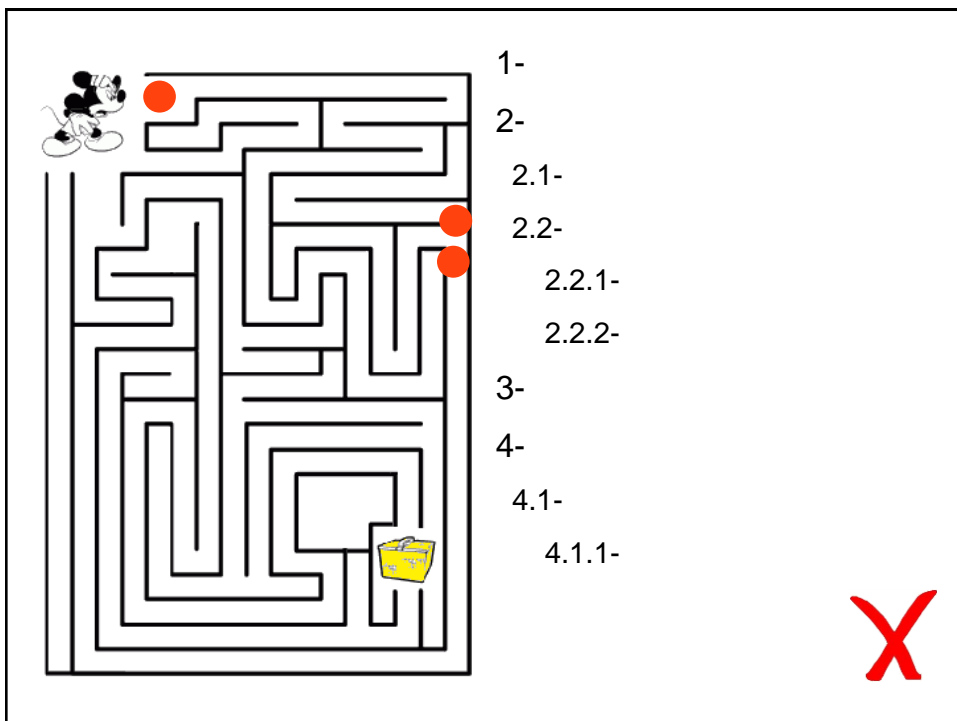
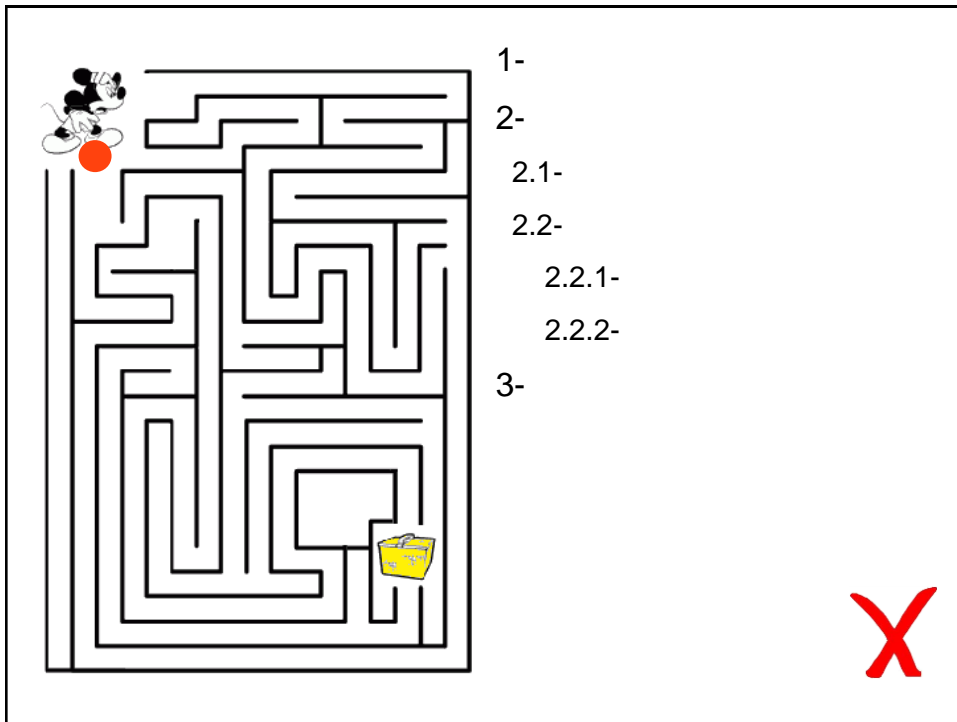
Tenemos varias **opciones**. Debemos tomar **decisiones** (¿cuál camino tomar?). Alguna(s) nos llevará(n) al éxito, otras al fracaso.

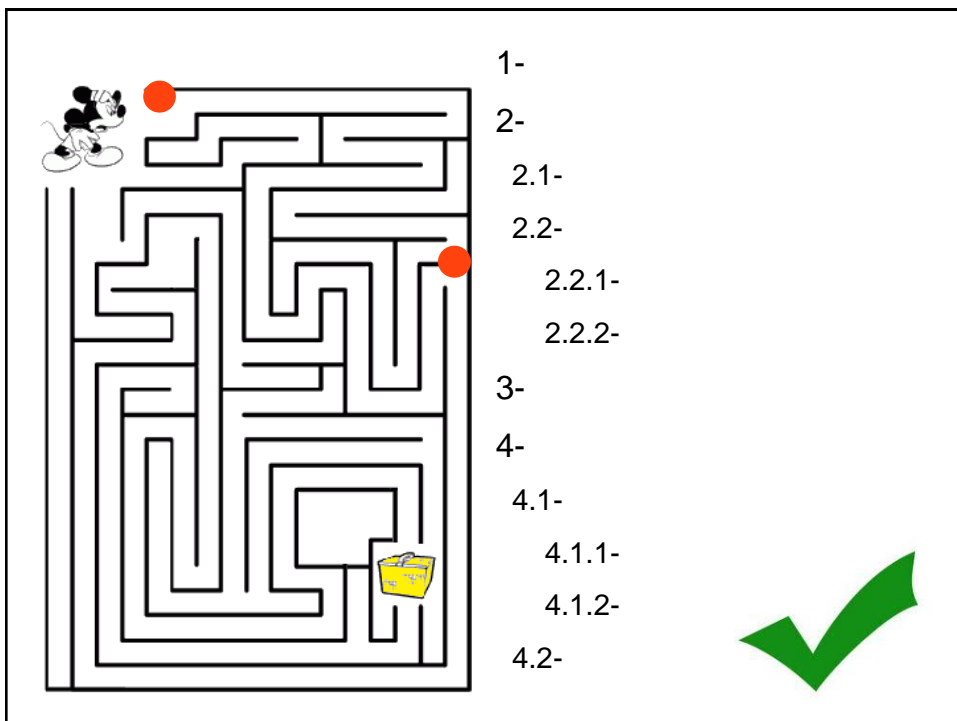
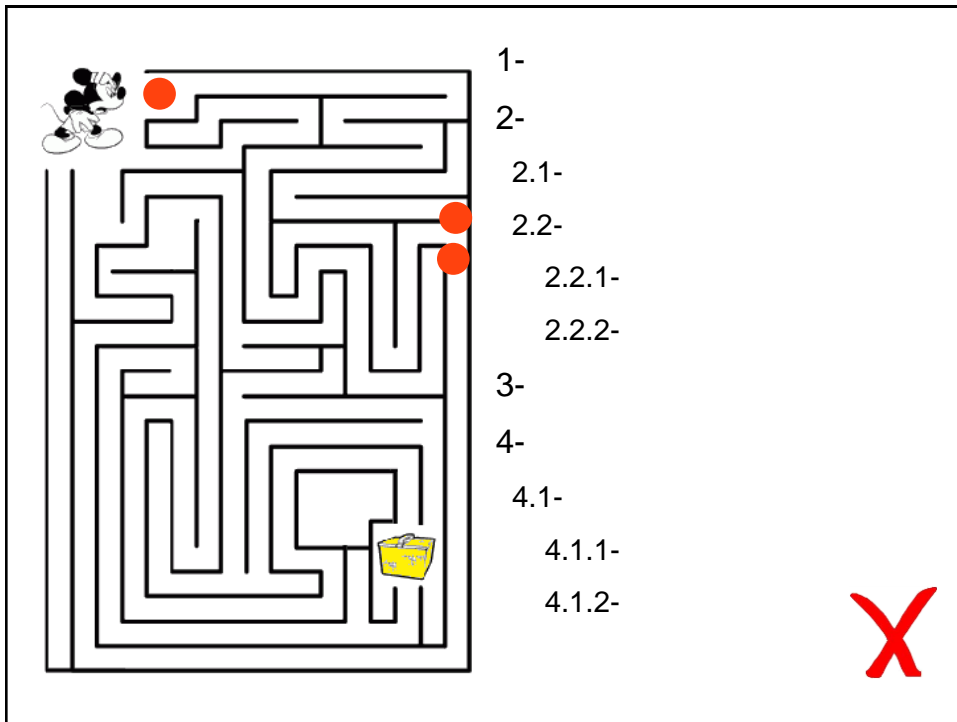


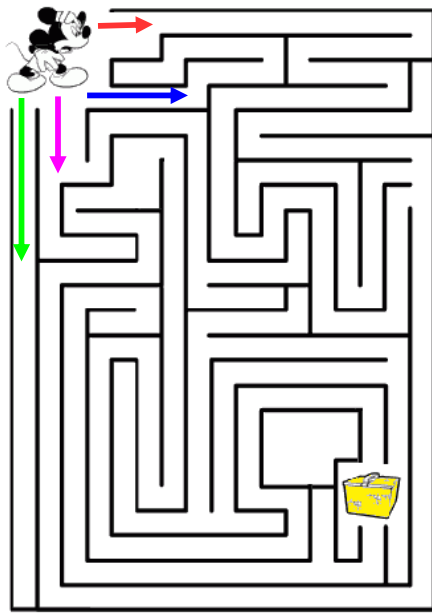
Solución:







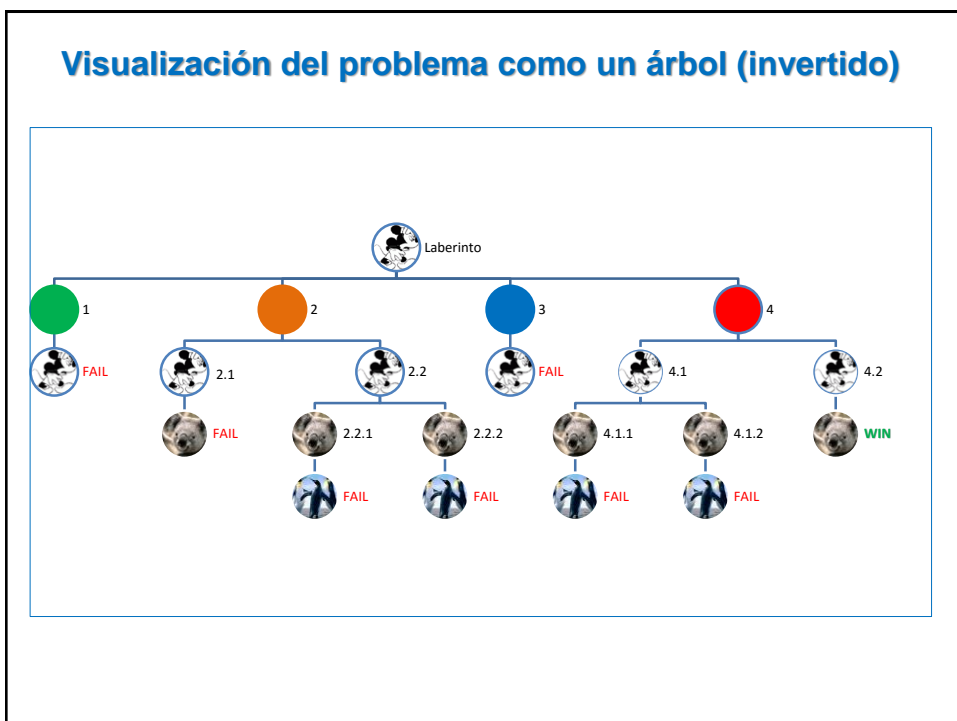




En este caso, el camino correcto fue el último que probamos.

Si el orden hubiese sido otro →

podríamos parar antes, si encontramos (por suerte!) la solución antes de revisar todos los caminos posibles.



¿Cómo diseñar la solución?

(para el laberinto de Mickey Mouse)

- Pensamos en una solución parcial. Por ejemplo, recorrer solo el primer camino y solo las primera alternativa de cada bifurcación.
- Definimos una función recursiva para esta solución parcial, asegurando de anotar el camino que recorremos. Es similar a la recursión para encontrar la sumatoria de un número N.
- Determinando la condición base de término. Esto es cuando llegamos al final del camino sea éste exitoso o no.
- Imprimimos el resultado del camino recorrido.
- Ahora generalizamos el algoritmo para recorrer todas las posibles alternativas, desde el primer nivel hasta el último.

Veamos ahora una solución, asumiendo que el laberinto de Mickey está definido como una lista de listas.

Laberinto de Mickey

```
""" Laberinto de Mickey Mouse """
def Laberinto(lab, camino=""):
    if len(lab) <= 1:
        if lab[0] == "F":
            resultado = " FAIL "
        else: resultado = " WIN!!! "
        print (camino+resultado)
        camino = ""
    else:
        for i in range(len(lab)):
            aux = camino + str(i+1)+"-->"
            Laberinto(lab[i], aux)

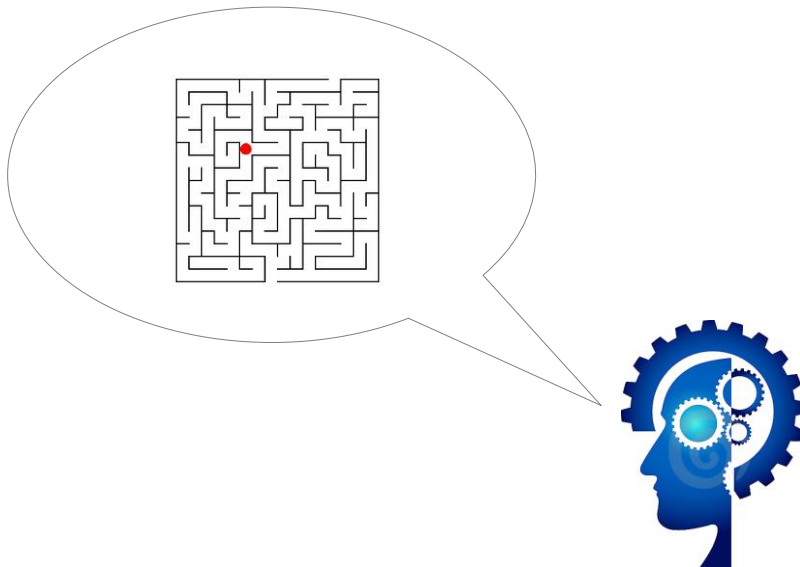
mickey = [["F"], [{"F"}, [{"F"}, {"F"}]], {"F"}, [{"F", "F"}, {"W"}]]
Laberinto(mickey)
```

Camino 1: ["F"]
 Camino 2: [{"F"}, [{"F"}, {"F"}]]
 Camino 3: ["F"]
 Camino 4: [{"F", "F"}, {"W"}]]

```
>>>
1--> FAIL
2-->1--> FAIL
2-->2-->1--> FAIL
2-->2-->2--> FAIL
3--> FAIL
4-->1-->1--> FAIL
4-->1-->2--> FAIL
4-->2--> WIN!!!
```

Idea general de backtracking (en pseudocódigo)

```
def backtracking (Solucion, solucionParcial):  
    if (noValeLaPenaSeguir(solucionParcial)):  
        terminar  
    if (sirve(solucionParcial)):  
        procesarSolucion(solucionParcial)  
    PosiblesSoluciones = []  
    PosiblesSoluciones = obtenerPosibilidades(solucionParcial)  
    for i in range (0, posiblesSoluciones.length):  
        backtracking (posiblesSoluciones[i])  
    hacerAlgoConLosResultadosYEntregarRespuesta()
```



Mini-Tarea 15 – Participación:

Escribe un programa recursivo-backtracking para resolver un laberinto, el cual se lee desde un archivo generado desde Excel (archivo .csv), columnas separadas por “;”.

Ejemplo de laberinto:

LABERINTO PROPUESTO:

```

X X X X X X X X X X X X X X X X X X
      X X X X      X
X X  X      X      X X X      X X
X      X X X X      X X X X      X X
X X X X X      X X X X X X      X X
X      X      X X X      X X X
X X X X X X X      X X X
X      X X X      X X X X X X
X X X X X X X X X X      X X X X
X      X      X X X X X X      X X
X X X X X X      X X X X      X
X      X      X X      X X X
X X X X X X X X X X X X X X X X X X

```

LABERINTO PROPUESTO:

```

X X X X X X X X X X X X X X X X
      X X X X      X
X X  X      X      X X X      X X
X      X X X X      X X X X      X X
X X X X X      X X X X X X      X X
X      X      X X X      X X X
X X X X X X X      X X X
X      X X X      X X X X X X
X X X X X X X X X X      X X X
X      X      X X X      X X X
X X X X X X X X X X X X X X X X

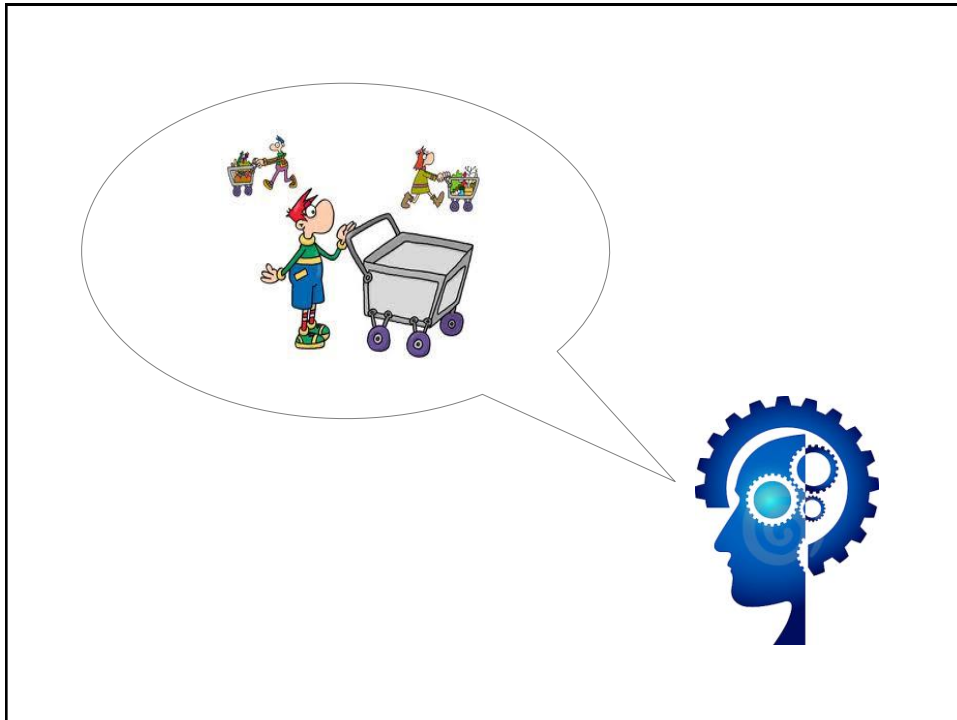
```

LABERINTO RESUELTO:

```

X X X X X X X X X X X X X X X X
* * * X X X X * * * X      * * * X
X X * X      X      * X * X      * X * X
X * *      X X X X * X * X X X * X * X
X * X X X X      X X * X * X X X * X * X
X *      X X X X * * * * * X * X * X
X * X X X X X X      X * X X      * * X
X * * * * * * * * * X X X X * X * X
X X X X X X X X X X      X X X * X * X
X      X      X X X X X X * X * X
X X X X X X      X X X X      * * X
X      X      X X      X X X *
X X X X X X X X X X X X X X X X

```

Problema Propuesto: Carro de supermercado

Es posible colocar los distintos productos de tal forma que solo un 10% de la capacidad del carro se pierde en los espacios entre un producto y otro.

¿Cómo determinar qué productos conviene colocar en el carro, tomando en cuenta sus volúmenes y precios asociados?

Hacer un programa que busque una configuración óptima tal que maximice el valor (precio) del contenido del carro. No interesa el detalle de los productos a elegir (pues quedan en el carro).

Asuma que posee dos listas ya inicializadas, `volumenes[]` y `precios[]`, con los volúmenes y precios de los productos, respectivamente (del mismo largo). También conocemos el valor `VC` que representa el volumen total del carro.

El arte de la recursión

Desafíos

- Las torres de hanoi
- Las 8 reinas (n-reinas)
- Movimientos de un caballo en el ajedrez

51



Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencias de la Computación

Introducción a la Programación IIC1103

Prof. Ignacio Casas
icasas@ing.puc.cl

Tema 10 – Recursión Parte 2

Colaboración de Mauricio Arriagada